SYSTEMS AND METHODS FOR VERIFYING LOCKSTEP OPERATION

BACKGROUND

Computer processor design is an extremely complex and lengthy process. The design process includes a range of tasks from high-level tasks such as specifying the architecture down to low-level tasks such as determining the physical placement of transistors on a silicon substrate. Each stage of the design process also involves extensive testing and verification of the design through that stage. One typical stage of processor design is to program the desired architecture for the processor using a register transfer language (RTL). The desired architecture is represented by an RTL specification that describes the behavior of the processor in terms of step-wise register contents. The RTL specification models what the processor does without describing the physical circuit details. Thus, the processor architecture can be verified at a high level with reference to the RTL specification, independent of implementation details such as circuit design and transistor layout. The RTL specification also facilitates later hardware design of the processor.

Manually verifying the RTL specification of the processor architecture is prohibitively complex during the design of a modern microprocessor. Therefore, multiple test cases are typically generated to test the design. Each test case contains input instructions and may also contain the desired results or outputs. Once created, the test cases may be executed on a simulation of the RTL specification (often compiled to increase speed) and the

results analyzed. Through that analysis, errors in the RTL specification, and potentially the processor architecture design, may be identified.

Many processors use multiple processor cores that execute instructions during processor operation. Cores of such processors are connected by an interface, such as a point-to-point (P2P) interface, typically on a single chip. With such a configuration, the processor may be operated in a "lockstep" mode in which two or more of the processor cores execute the same instruction stream each clock cycle. Given that the behavior of the cores is deterministic, the same output should result from each processor core operating in lockstep mode. One advantage of operating in lockstep mode is that if one of the cores experiences an error (e.g., a manufacturing defect, a stuck-at fault, a soft error from an alpha particle, a transient electrical failure, etc.), the other core(s), at least in theory, can continue to execute so that the processor can continue to operate. Assuming that the core that experienced the error has not failed completely, the operating system may be able to resynchronize that core so as to resume normal lockstep operation. In cases in which the cores of a processor are configured to operate in lockstep mode, those cores are typically connected to a lockstep block that monitors the operation of the cores and identifies certain observed errors when they arise.

Currently, no automated systems or methods for verifying lockstep block operation, and therefore processor lockstep operation, are known.

SUMMARY

Disclosed are systems and methods for verifying lockstep operation. In one embodiment, a system and a method pertain to monitoring interface signals, detecting output of a modeled lockstep block, comparing the detected output with an expected output for the

lockstep block relative to a current modeled machine state, and flagging a lockstep block error if the detected output does not match the expected output.

BRIEF DESCRIPTION OF THE DRAWINGS

The disclosed systems and methods can be better understood with reference to the following drawings. The components in the drawings are not necessarily to scale.

FIG. 1 is a block diagram of an embodiment of a system for verifying a processor architecture.

FIG. 2 is a block diagram illustrating an example of logical data flow in a point-to-point link network.

FIGs. 3A and 3B comprise a flow diagram of an embodiment of a method for verifying lockstep operation.

FIG. 4 is a flow diagram of an embodiment of a method for verifying lockstep operation.

FIG. 5 is a block diagram of an embodiment of a computer system in which lockstep operation may be verified.

DETAILED DESCRIPTION

Disclosed are systems and methods for verifying lockstep operation. Referring to FIG. 1, a processor architecture verification system 1 is illustrated that verifies processor architecture by executing at least one test case 10 on both a register transfer language (RTL) simulator 12 that comprises a compiled version of the RTL specification, and a golden simulator 14 that comprises a relatively high-level program that emulates operation of the processor. It is noted that the golden simulator 14 is not required for lockstep operation verification. The golden simulator 14 is shown and identified herein, however, in that it may

optionally be utilized in the lockstep operation verification process and may be useful for other aspects of processor architecture verification beyond lockstep operation.

The RTL simulator 12 and the golden simulator 14 both simulate the desired processor architecture 16 and 18, respectively. The RTL simulator 12 and the golden simulator 14 may, however, comprise different output interfaces. For instance, the RTL simulator 12 may comprise a point-to-point (P2P) link network output interface while the golden simulator 14 may comprise a front side bus (FSB) output interface. As is described in greater detail below, the modeled architecture 16 includes multiple processor cores that enable lockstep operation, and a lockstep block that monitors the operation of the cores to identify certain errors in core operation when they arise.

Because the output of the RTL simulator 12 and the golden simulator 14 may be in different formats, a translator 22 may be provided that translates the output of the RTL simulator to match the format of the golden simulator 14. The translated output of the RTL simulator 12 can then be compared with the output of the golden simulator 14 in a comparator 20 to produce test results 28. In the illustrated embodiment, the comparator 20 comprises part of the golden simulator 14. Alternatively, however, the comparator 20 may be independent of the golden simulator 14. If any differences in the outputs are detected by the comparator 20, the processor designer is alerted to the fact that an error may exist in the RTL simulator 12 or the golden simulator 14 or both. This enables test cases to be applied to the processor architecture quickly while minimizing required designer attention.

In some embodiments, the translator 22 de-pipelines the output of the RTL simulator 12 for comparison with the output of the golden simulator 14. In such an embodiment, the translator 22 may be referred to as a "depiper". Such de-pipelining may be necessary because the golden simulator 14 is typically more abstract than the RTL simulator 12. For instance, the golden simulator 14 may not include the same level of detail about the processor

architecture being verified as does the RTL simulator 12. The result is that the output of the RTL simulator 12 may not directly match the output of the golden simulator 14 even though the underlying architecture 16, 18 is the same and the test case 10 is identical. A detailed example of a suitable depiper is described in U.S. Patent 5,404,496, which is incorporated by reference herein for all that it discloses.

In the embodiment shown in FIG. 1, the translator 22 comprises a virtual bus interface (VBI) 24 that translates transactions from the RTL simulator 12 from P2P link network format to FSB format for comparison with the FSB format output of the golden simulator 14. In addition to the VBI 24, the translator 22 comprises a lockstep block checker 26 that, as is described in greater detail below, monitors the operation of multiple processor cores (modeled in the architecture 16) as well as the lockstep block when the modeled processor operates in the lockstep mode. Although the lockstep block checker 26 is shown as comprising part of the translator 22 (e.g., depiper), it is noted that the lockstep block checker may be located anywhere (including independent of the translator) in which it may monitor the operation of processor cores and lockstep block during lockstep mode operation. In most embodiments, however, the checker 26 is implemented independent of the golden simulator 14 both to avoid the complexity associated therewith and due to the fact that the golden simulator 14 may be too high level to evaluate (or even be aware of) lockstep operation. In such cases, the lockstep block checker 26 may adjust the output (e.g., state-update packets) so as to fool the golden simulator 14 into "thinking" that only one processor core is running when more than one such core is operating in lockstep mode.

The RTL simulator 12 and the golden simulator 14 are operated relative to information specified by the test case 10. By way of example, the test case 10 comprises a program to be executed on the processor architecture 16 and 18 in the RTL simulator 12 and golden simulator 14, respectively. The test case program is a memory image of one or more

computer executable instructions, along with an indication of the starting point, and may comprise other state specifiers such as initial register contents, external interrupt state, etc. Accordingly, the test case 10 defines an initial state for the processor that is being simulated and the environment in which it operates. The test case 10 may be provided for execution on the RTL simulator 12 and golden simulator 14 in any suitable manner, such as an input stream or an input file specified on a command line.

The RTL specification used to generate the RTL simulator 12 may be implemented using any suitable tool for modeling the processor architecture 16, such as any register transfer language description of the architecture, which may be interpreted or compiled to act as a simulation of the processor. The RTL simulator 12 of an exemplary embodiment contains an application program interface (API) that enables external programs, including the translator 22, to access the state of various signals in the simulated processor such as register contents, input/outputs (I/Os), etc. Thus, the output of the RTL simulator 12 may be produced in any of a number of ways, such as an output stream, an output file, or as states that are probed by an external program through the API. The RTL simulator 12 may simulate any desired level of architectural detail, such as the processor cores, or the processor cores and one or more output interfaces.

As noted above, the golden simulator 14, when provided, is a relatively abstract, higher-level simulation of the processor architecture, and therefore may be less likely to include faults or errors than the RTL simulator 12. The golden simulator 14 is written using a high-level programming language such as C or C++. Alternatively, the golden simulator 14 may be written using any other suitable programming language, whether compiled, interpreted, or otherwise executed. Whereas the RTL simulator 12 actually matches the details and reality of the processor being simulated to a great degree, the golden simulator 14

typically is a conceptual model without concern for timing considerations arising from physical constraints.

The translator 22 (e.g., depiper) tracks instructions as they flow through the RTL simulator 12 and notes their effects on the simulated processor. The translator 22 may generate a retire record for each instruction that indicates when the instruction started executing and when it completed or retired, along with the states that changed during execution. In some cases, if state changes cannot be tracked to a single instruction, the depiper may generate a generic report identifying an altered state and the instructions that may have caused the change.

In some embodiments in which the translator 22 comprises a depiper, the VBI 24 works in parallel with the depiper, with the depiper producing state change records such as depiper retire records, and the VBI producing state change records in the form of synthesized FSB transactions. Although the VBI 24 may read the P2P packets directly from the P2P interface on the RTL simulator 12 and may access information about the RTL simulated processor via the API, the VBI may also access information about the RTL simulated processor that is stored in the depiper. In some embodiments, the depiper contains structures that monitor the simulated processor cores' states. In such cases, it may be convenient for the VBI 24 to access some information from the depiper for use in reporting or synthesizing fields used in the FSB phases.

In some embodiments in which the translator 22 comprises a depiper, the depiper first reads the P2P output of the RTL simulator 12 and de-pipelines the P2P transactions, generating a de-pipelined version of the P2P transactions. The VBI 24 then reads the de-pipelined version of the P2P transactions from the depiper and generates corresponding FSB transactions for the comparator 20. The de-pipelined P2P transactions may be transferred

from the depiper to the VBI 24 in any suitable manner, such as across a virtual P2P link or in a file containing depiper retire records.

Notably, the VBI 24 is not limited to use with verification systems including a depiper. Verification systems having the same level of pipelining detail in both the RTL simulator 12 and the golden simulator 14 may not need a depiper, but a VBI 24 still enables processor simulators with different output interfaces to be used together. If the translator 22 comprises a depiper, the VBI 24 may access information stored in the depiper as described above, or may be implemented as a module in the depiper for convenience. In embodiments in which the translator 22 does not include a depiper, the VBI 24 in the translator still directly connects to the P2P output of the RTL simulator 12, but obtains other information about the state of the simulated processor from the RTL simulator via the API. The VBI 24 uses the resulting P2P packets and other information to produce translated FSB transactions in whatever manner required by the comparator 20, such as generating a virtual FSB connection to the comparator, or generating output reports containing records of FSB format transactions that may be read by the comparator.

FIG. 2 illustrates an example output interface of the RTL simulator 12. As shown in that figure, the RTL simulator 12 uses one or more ports into a point-to-point (P2P) link network 30 shown in FIG. 2. The P2P link network 30 is a switch-based network with one or more crossbars 32 acting as switches between components such as processor cores 34 (i.e., Core 1 and Core 2 in the embodiment of FIG. 2), memory 36, or other devices (not shown). Transactions are directed to specific components and are appropriately routed in the P2P link network 30 by the crossbar 32. The routing provided by the crossbar 32 reduces the load on the system components because they do not need to examine each broadcast block of information. Instead, each component ideally receives only data meant for that component. Use of the crossbar 32 also avoids bus loading issues, thereby facilitating scalability.

Transactions on the P2P link network 30 are packet-based, with each packet containing a header comprising routing and other information. Packets containing requests, responses, and data are multiplexed so that portions of various transactions may be executed with many others at the same time. Transmissions are length limited, with each length-limited block of data called a "flit." Thus, a long packet will be broken into several flits, and transactions will typically require multiple packets. Therefore, the P2P link network 30 is monitored over time to collect the appropriate P2P packets until enough information exists for a corresponding FSB phase to be generated by the translator 22. To achieve such monitoring, the translator 22 monitors a port 42 on the crossbar 32 that is connected to the cores 34 in the RTL simulator 12. An exemplary read operation in a P2P link network is described in U.S. Patent Application Serial No. 10/700,288 (attorney docket number 200209129-1), filed November 3, 2003, which is incorporated herein for all that it discloses.

As is further illustrated in FIG. 2, the RTL simulator 12 includes a lockstep block 38 that resides between the processor cores 34 and their respective core protocol engines (CPEs) 40. The lockstep block 38 monitors outputs of the modeled processor cores 34 (i.e., Core 1 and Core 2 in the embodiment of FIG. 2) to identify when core errors occur. Such errors typically come in two main types. The first type of error comprises an error that the cores 34 detect, i.e., self-detected errors. In such cases, the core 34 experiencing the error (i.e., the failing core) outputs an error message that is intercepted by the lockstep block 38, and the lockstep block ensures that no data from the failing core is output from the processor. In addition, the lockstep block 38 issues a system-level alert that signifies that the failed core must be resurrected to resume lockstep operation.

The other main type of error occurs when no error is detected by a processor core, but different data is output from the cores that are operating in lockstep mode. As noted above, the outputs from the cores should be identical in that the cores' behavior is deterministic and

because the cores execute the same instruction streams. Accordingly, when different outputs are detected by the lockstep block 38, one or more of the cores is experiencing an error. In such as case, the lockstep block 38 raises a system-wide error on the interface and further execution is halted and neither core is allowed to send data to the system to prevent system data corruption in that it is not known which of the cores is failing and which is operating correctly.

As noted above, it is desirable to analyze the lockstep block's behavior to properly verify a design of a processor. In the embodiments described herein, the operation of the lockstep block 38 can be monitored and analyzed using the lockstep block checker 26. The lockstep block checker 26 implements a software model of the lockstep state machine that describes the proper operation the lockstep block 38 in various system states, and monitors the RTL simulator 12 signals that are output from the cores and that are input into and output out of the lockstep block. From those interface signals, the lockstep block checker 26 can evaluate the operation of the lockstep block 38 and identify errors in that operation when applicable. Such an error identifies a potential flaw in the design of the physical lockstep block that will be used in the actual processor.

FIG. 3 provides an example embodiment of verifying lockstep operation and, more particularly, of verifying operation of a lockstep block using the lockstep block checker 26. In this example, it is presumed that the system is operating in lockstep mode. By way of example, the flow described in the following is performed once during each clock tick. Beginning with block 300 of FIG. 3, the lockstep block checker 26 monitors the interface (e.g., the P2P interface 30) and captures interface signals that are issued on that interface. Such monitoring is possible in that, because the translator 22 (e.g., depiper) monitors each channel of the P2P interface, the lockstep block checker 26 can access all traffic that is transmitted over the interface. With reference to decision block 302, it can be determined if

an error signal is output by a processor core (e.g., Core 1 or Core 2). Such an error signal results from self-detected errors of the cores. If no such error signal is detected by the lockstep block checker 26, flow continues to block 318 of FIG. 3B, which is described below. However, if such an error signal is detected, flow continues to block 304 at which the lockstep block checker 26 transitions its state machine model into a core-disabled mode.

Once the state machine model has been transitioned into the core-disabled mode, the lockstep block checker 26 examines the output error signal(s) of the lockstep block, as indicated in block 306, to determine whether that/those signal(s) fired at an expected time. The expected time is determined by the lockstep block checker 26 using its knowledge of the lockstep block as well as the inputs into the lockstep block. Specifically, in that the configuration and mode of operation of the lockstep block is known (from the state machine model), the lockstep block checker 26 can determine from the inputs into the lockstep block and the time at which those inputs were received by the lockstep block what error signal(s) should be issued by the lockstep block and when. By way of example, the actual process of determining the expected signals and times may comprise accessing a data structure, such as a table, that cross-references input signals (to the lockstep block) with the output signals (from the lockstep block) that should result from the input signals, as well as the times at which the output signals should be output. Alternatively, expected times can be calculated using an appropriate algorithm that has as inputs the input signals and the times at which they were received by the lockstep block. In either case, the time at which an expected signal is expected to fire can be scheduled and the interface can be monitored for those signals.

With reference to decision block 308, if the error signal(s) is/are not fired at the expected time(s), the lockstep block behavior is incorrect and, as indicated in block 310, the lockstep block checker 26 flags a lockstep block error to signal that a problem exists with the lockstep block design (or with the way in which the design has been modeled). Once such an

error has been detected and flagged, further testing of the processor architecture may either cease or continue. For the purposes of this example, however, it is assumed that the occurrence of such an error causes testing to cease, in which case flow for the session is terminated (see reference B in FIGs. 3A and 3B).

With reference back to decision block 308, if the error signal(s) is/are fired at the expected time(s), the lockstep block reacted appropriately in relation to the error signal output by the failing core. In such a case, flow continues to block 312 at which the data values output by the "healthy" core(s), i.e., the core(s) that did not output the error signal, are compared with the data output of the lockstep block (i.e., data enroute to a CPE 40). Again, given that the lockstep block checker 26 knows the configuration of the lockstep block and the manner in which the block is supposed to operate, the lockstep block checker can determine the proper output of the lockstep block based upon the input provided to the block (i.e., the output from the healthy core(s)). With reference to decision block 314, if the values output from the lockstep block differ from the values that the lockstep block checker 26 is expecting, the lockstep block checker assumes that the lockstep block is not functioning properly and, therefore, flags a lockstep block error, as indicated in block 316. Again, flow may then terminate at that point.

If the values output by the lockstep block match those expected by the lockstep block checker 26 in decision block 314, or if no error signal was output by a core in decision block 302, flow continues to block 318 of FIG. 3B. As indicated in that block, the lockstep block checker 26 next inputs the captured values (see block 300 of FIG. 3A) into its state machine model. Through such input, the lockstep block checker 26 can compare the data values from each lockstep core, as indicated in block 320, so that the checker can determine whether the cores are producing the same outputs, in which case they are assumed to be working

properly, or producing different outputs, in which case at least one of the cores is failing. By way of example, this comparison can be conducted using an XOR tree.

With reference next to decision block 322, if different values are not observed by the lockstep block checker 26, flow reverts back to block 300 of FIG. 3A at which monitoring and the flow described above resumes. By way of example, such flow may occur during the next clock tick. If, on the other hand, different values are observed, flow continues to block 324 at which the lockstep block checker 26 transitions the state machine model into a difference-detected mode. Once the state machine model is transitioned into that mode, the lockstep block checker 26 examines the fatal error output signal(s) (e.g., BINIT signals) from the lockstep block, as indicated in block 326. In particular, the lockstep block checker 26 determines, from the outputs of the cores, when such signals are expected. Therefore, with reference to decision block 328, the lockstep block checker 26 can determine whether the signal(s) fired at the expected time. If so, the lockstep block has performed correctly and flow can return to block 300 of FIG. 3A. If not, however, the lockstep block has operated incorrectly and, therefore, the lockstep block checker 26 flags a lockstep block error, as indicated in block 330.

In view of the above, a method for verifying lockstep operation may be as provided in FIG. 4. With reference to that figure, the method comprises monitoring interface signals (400), detecting output of a modeled lockstep block (402), comparing the detected output with an expected output for the lockstep block relative to a current modeled machine state (404), and flagging a lockstep error if the detected output does not match the expected output (406).

FIG. 5 is a block diagram of a computer system 500 in which the foregoing systems can execute and, therefore, a method for verifying lockstep operation can be practiced. As indicated in FIG. 1, the computer system 500 includes a processing device 502, memory 504,

at least one user interface device 506, and at least one input/output (I/O) device 508, each of which is connected to a local interface 510.

The processing device 502 can include a central processing unit (CPU) or an auxiliary processor among several processors associated with the computer system 500, or a semiconductor-based microprocessor (in the form of a microchip). The memory 504 includes any one or a combination of volatile memory elements (e.g., RAM) and nonvolatile memory elements (e.g., read only memory (ROM), hard disk, etc.).

The user interface device(s) 506 comprise the physical components with which a user interacts with the computer system 500, such as a keyboard and mouse. The one or more I/O devices 508 are adapted to facilitate communication with other devices. By way of example, the I/O devices 508 include one or more of a universal serial bus (USB), a Firewire, or a small computer system interface (SCSI) connection component and/or network communication components such as a modem or a network card.

The memory 504 comprises various programs including an operating system 512 that controls the execution of other programs and provides scheduling, input-output control, file and data management, memory management, and communication control and related services. In addition to the operating system 512, the memory 504 comprises the RTL simulator 12 and the translator 22 identified in FIG. 1. As is shown in FIG. 5, the translator 22 includes the VBI 24 and the lockstep block checker 26, which have been described in detail above.

Various programs (i.e., logic) have been described herein. Those programs can be stored on any computer-readable medium for use by or in connection with any computer-related system or method. In the context of this document, a computer-readable medium is an electronic, magnetic, optical, or other physical device or means that contains or stores a computer program for use by or in connection with a computer-related system or method.

These programs can be embodied in any computer-readable medium for use by or in connection with an instruction execution system, apparatus, or device, such as a computer-based system, processor-containing system, or other system that can fetch the instructions from the instruction execution system, apparatus, or device and execute the instructions.